aiorethink Documentation

Release

Lars Tiede

June 19, 2016

Contents

aiorethink is a fairly comprehensive but easy-to-use asyncio-enabled Object Document Mapper for RethinkDB. It is currently in development.

Source: https://github.com/lars-tiede/aiorethink

Contents:

CHAPTER 1

Installation

aiorethink is available on PyPI, so you can install it simply with pip. You can also grab the source right from github.

1.1 Prerequisites

You need at least Python 3.5. Check the version you have like so:

```
python3 --version
```

Along with Python 3.5, you should also have pip.

Obviously, you also need access to an instance of RethinkDB.

1.2 Alternative 1: install from PyPI

```
pip3 install aiorethink
```

1.3 Alternative 2: install from source

Running or installing right from source gives you the possibility to run the bleeding edge version of aiorethink. There is also more content for you to play with. For instance, the sources include a Vagrantfile, which gives you a VM running RethinkDB and all you need for playing with aiorethink.

The easiest way to install aiorethink from source is to use pip to install aiorethink right from github. To do this in a fresh virtual environment, run this in an empty directory:

```
python3 -m venv py-env
. ./py-env/bin/activate
pip3 install -e git+https://github.com/lars-tiede/aiorethink@master#egg=aiorethink
```

You now have aiorethink installed locally. Since we specified `-e` ("editable mode") to pip, the cloned repository will be stored in `py-env/src/aiorethink`. You can hack things in there if you want. If you don't need the repository clone, just omit `-e`.

You can use any version of aiorethink this way, by specifying any git branch, tag, or commit instead of `master`.

1.4 Alternative 3: hack and build from source

If you don't want the cloned and hackable repository to live in `py-env/src/aiorethink`, you can also clone the repository yourself and make an environment that's more tailored towards hacking aiorethink specifically.

Clone the repository:

```
git clone https://github.com/lars-tiede/aiorethink
```

Then cd into the aiorethink directory. Make a virtualenv and install everything you need into it like so:

```
python3 -m venv py-env
. ./py-env/bin/activate
pip3 install -r requirements.txt
```

If you want to run the test suite and build an aiorethink package, install the pip packages from the other requirements files as well:

```
pip3 install -r requirements-test.txt -r requirements-dev.txt
```

Now you can run a python interpreter and `import aiorethink` and start using aiorethink.

If you want to build an aiorethink distribution package (a "wheel") that you can install with pip somewhere else (say, your own project's virtualenv), do this:

```
python setup.py bdist_wheel
```

You'll have a wheel file in the `build` directory now. You can install the wheel somewhere else by pointing pip right to the file:

```
pip install PATH_TO_WHEEL_FILE
```

CHAPTER 2

API Reference

2.1 FieldContainer and Document

aiorethink.FieldContainer and *aiorethink.Document* store declared `aiorethink.Field` objects and undeclared data.

FieldContainers can be used everywhere where declared data is used, for instance as named fields of other FieldContainers or as items of a list.

Documents are FieldContainers with logic for saving them to, and loading them from, a database. Unlike FieldContainers, Documents are required to be "top level" objects, i.e. you can not store a Document object inside of another Document object. For nesting, use FieldContainers. For referencing other documents, use lazy references.

class `aiorethink.`**`FieldContainer`**(***kwargs*)

Bases: `collections.abc.MutableMapping`

A FieldContainer stores named fields. It is the base for *aiorethink.Document*, but can also be used directly.

There are declared fields (i.e. instances of Field), and there are undeclared fields. Both can be accessed using a dict-like interface. Only declared fields can be accessed by attribute.

A FieldContainer can be stored to and loaded from a RethinkDB. This way, it can act as a "value". An associated ValueType class, FieldContainerValueType, makes it possible to use FieldContainer values "anywhere".

`clear`(*which=0*)

Deletes all fields.

`copy`(*which=0*)

Creates a new FieldContainer (same class as self) and (shallow) copies all fields. The new FieldContainer is returned.

`dbitems`(*which=0*)

Returns ItemsView of (db_key, db_value).

`dbkeys`(*which=0*)

Returns a KeysView of database field names.

`dbvalues`(*which=0*)

Returns a ValuesView of suited-for-DB representations of values.

classmethod `from_cursor`(*cursor*)

Returns an `aiorethink.db.CursorAsyncMap` object, i.e. an asynchronous iterator that iterates

over all objects in the RethinkDB cursor. Each object from the cursor is loaded into a FieldContainer instance using `cls.from_doc`, so make sure that the query you use to make the cursor returns "complete" FieldContainers with all its fields included.

Usage example:

```
# TODO make an example for the more general FieldContainer
conn = await aiorethink.db_conn
all_docs_cursor = MyDocument.cq().run(conn)
async for doc in MyDocument.from_cursor(all_docs_cursor):
    assert isinstance(doc, MyDocument) # holds
```

classmethod `from_query` (*query*, *conn=None*)

First executes a ReQL query, and then, depending on the query, returns either no object (empty result), one FieldContainer object (query returns an object), or an asynchronous iterator over FieldContainer objects (query returns a sequence). When writing your query, you know whether it will return a single object or a sequence (which might contain one object).

The query may or may not already have called run(): * if run() has been called, then the query (strictly speaking, the

awaitable) is just awaited. This gives the caller the opportunity to customize the run() call.

•if run() has not been called, then the query is run on the given connection (or the default connection). This is more convenient for the caller than the former version.

If the query returns None, then the method returns None.

If the query returns an object, then the method loads a FieldContainer object (of type `cls`) from the result. (NB make sure that your query returns the whole container, i.e. all its fields). The loaded FieldContainer instance is returned.

If the query returns a cursor, then the method calls `cls.from_cursor` and returns its result (i.e. an `aiorethink.db.CursorAsyncMap` object, an asynchronous iterator over FieldContainer objects).

`get_dbvalue` (*fld_name*, *default=None*)

Returns suitable-for-DB representation (something JSON serilizable) of the given field. If the field is a declared field, some conversion might be involved, depending on the field's value type. If the field exists but is undeclared, the field's value is returned without any conversion, even if that is not json serializable. If the field does not exist, default is returned.

`keys` (*which=0*)

Returns a KeysView of field names.

`set_dbvalue` (*fld_name*, *dbvalue*, *mark_updated=True*)

Sets a field's 'DB representation' value. If fld_name is not a declared field, this is the same as the 'python world' value, and set_dbvalue does the same as __setitem__. If fld_name is a declared field, the field's 'python world' value is constructed from dbvalue according to the field's value type's implementation, as if the field is loaded from the database.

Note than *fld_name* refers to the field's "document field name", which might be different from the field's "database field name". You can convert a "database field name" to a "Document field name" using `get_key_for_dbkey()`.

`to_doc`()

Returns suited-for-DB representation of the FieldContainer.

`validate`()

Explicitly validate the field container. aiorethink does this automatically when necessary (for example when an Document is saved).

The default implementation validates all updated fields individually.

When you override this, don't forget to call super().validate().

The method returns self.

validate_field(*fld_name*)
Explicitly validate the field with the given name. This happens automatically for most fields when they are updated. The exception to this are fields where aiorethink can not know when updates happen, for example when you change an element within a list.

The method returns self.

values(*which=0*)
Returns a ValuesView of values.

class aiorethink.**Document**(***kwargs*)
Bases: aiorethink.values_and_valuetypes.field_container.FieldContainer

Non-obvious customization: cls._table_create_options dict with extra kwargs for rethinkdb.table_create

aiter_changes(*conn=None*)
Note: be careful what you wish for. The document object is updated in place when you iterate. Unsaved changes to it might then be overwritten.

classmethod aiter_table_changes(*changes_query=None*, *conn=None*)
Executes *changes_query* and returns an asynchronous iterator (a ChangesAsyncMap) that yields (document object, changefeed message) tuples.

If *changes_query* is None, *cls.cq().changes(include_types = True)* will be used, so the iterator will yield all new or changed documents in cls's table, and changefeed messages will have a "type" attribute giving you more information about what kind of change happened.

If you sepcify *changes_query*, the query must return one complete document in new_val on each message. So don't use pluck() or something to that effect in your query.

The query may or may not already have called run(): * if run() has been called, then the query (strictly speaking, the

awaitable) is just awaited. This gives the caller the opportunity to customize the run() call.

•if run() has not been called, then the query is run on the given connection (or the default connection). This is more convenient for the caller than the former version.

copy(*which=0*)
Creates a new Document (same class as self) and (shallow) copies all fields except for the primary key field, which remains unset. The new Document is returned.

classmethod cq()
RethinkDB query prefix for queries on the Document's DB table.

classmethod create(***kwargs*)
Makes a Document and saves it into the DB. Use keyword arguments for fields.

classmethod load(*pkey_val*, *conn=None*)
Loads an object from the database, using its primary key for identification.

q()
RethinkDB query prefix for queries on the document.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

a

A

C

D

F

G

K

L

Q

S

T

V